Konstantin Mishin

Low Power Firmware Design in Embedded Systems

Metropolia University of Applied Sciences

Bachelor of Engineering

Electronics

Thesis

9 May 2017

Metropolia

Author	Konstantin Mishin
Title	Low Power Firmware Design in Embedded Systems
Number of Pages	38 pages + 2 appendices
Date	9 May 2017
Degree	Bachelor of Engineering
Degree Program	Degree Programme in Electronics
Professional Major	Embedded Systems
Instructors	Kalle Pokki, Project Manager Janne Mäntykoski, Senior Lecturer

This thesis analyzes a low-level embedded C-based approach to firmware design for a new I/O device that is to serve as an interface for an existing sensor array. A strong emphasis was given to investigating ways to reduce device power consumption while maintaining the same level of functionality.

The system implemented communications by USART with DMA in an interrupt-driven program flow, centered around an external input interrupt that recurred every 25ms. All hardware initializations and processes were implemented with register-level code, following the MCU reference manual closely.

The required functionality was achieved successfully, and the resulting device fulfills all the demanded functionality and somewhat more: the projected characteristics of the device allowed for application of many power saving methods, e.g running the entire program from RAM, and lowering system clocks to the kHz range. A mechanism that automatically corrects and recovers from communication errors was also implemented, and thus the device is more robust and may run for long periods of time reliably.

Keywords	Embedded, C, ARM Cortex M0, M0+, USART, DMA

Metropolia

Contents

List of Abbreviations

CPU	Central processing unit.
MCU	Microcontroller unit.
ADC	Analog to digital converter.
DAC	Digital to analog converter.
USART	Universal synchronous asynchronous receiver-transmitter
DMA	Direct Memory Access.
ISR	Interrupt service request.
NVIC	Nested Vector Interrupt Controller
NVM	Non-Volatile Memory
RAM	Random Access Memory
AMBA	Advanced Microcontroller Bus Architecture
AHB	AMBA High-Performance Bus

1 Introduction

An embedded system, put simply, is an applied computer system – distinct from general computing devices such as personal computers or smart devices by the fact that it is dedicated to a single function, or a narrow spectrum of functions. The system is called an embedded system because it is embedded within a larger electrical or mechanical system, and can fulfill various functions within it. [1,6]

A major part of what constitutes a complete embedded system is firmware – software that exerts near-complete low-level control of device functionality. Firmware is typically stored on the ROM – flash memory, for example [1, 618]. In more complex devices, firmware provides a standard operating environment for more sophisticated software[2,1], however, in simpler devices, it often constitutes the entirety of the operating system. The device at hand in this project is the latter.

The goal of this project is to create device firmware for an add-in board that communicates with a larger, existing and in-production master device. The firmware will be built with an emphasis on low power consumption: it will be the requirement that decides what features are to be used, and to a certain degree, it therefore decides overall device performance.

As the firmware is to be made from scratch and no existing codebase for this device needs to be considered, the intention is to implement the required functionality in the most programmatically efficient way possible, utilizing programming techniques that allow working with the hardware closely, with very little overhead or abstraction.

The purpose of taking the above approach over any other intends to investigate the possible benefits that a low-level programming approach may yield to device performance, *especially* in the scope of reducing power consumption, as that is a significant trend in modern embedded systems.

2 Embedded Systems

2.1 Introduction to Embedded Systems

2.1.1 Characteristics of Embedded Systems

“Embedded systems” is a generic term that encompasses a wide range of applied computer hardware that, on the surface, has little in common: a hearing aid, missile guidance system, and railway signal system are all embedded systems. Despite the apparent dissonance however, there are two defining characteristics that all these systems share:

1.) Real-world interaction: All embedded systems interact with the real, physical world: controlling some specific hardware, receiving signals through *sensors*, sending output signals to *actors* that somehow manipulate the environment. That environment, including the actors and sensors, is often referred to as the *plant*. [3, 5].

2.) Dedication to a limited set of functions: Most embedded systems are primarily designed to fulfill one specific function, or a specific set of functions, and are not fit for general-purpose computing [1, 5]

These two characteristics can be applied to the majority of embedded systems, with the exception of items such as smartphones or web pads, two lines of products whose classification as an embedded system is a topic of debate [1, 6].

There are also other common characteristics which are applicable to specific embedded systems, and therefore, not universal:

3.) A limited pool of hardware resources. In practice, that means less processing power, memory, I/O capabilities, power consumption – and simpler software, generally tailored to the application at hand. [1, 5]

4.) A high level of component integration. Embedded systems are commonly implemented around a MCU, which contains most necessary functionality – processor

core, memory, and I/O within a single package. That reduces overall system cost and *potentially* reduces overall system power consumption. [1, 129-130]

5.) Robustness and real-time constraints. As many embedded systems are implemented within mission critical applications (such as control of a safety system in a car, or a regulator in a power supply), an unambiguous, fast (frequently real-time) and consistent system response is key. [1, 6], [2,13]

2.1.2 Structure of an Embedded System:

The structure of a typical embedded system can be generalized with relative ease. The following figure presents a *minimal* embedded system configuration:

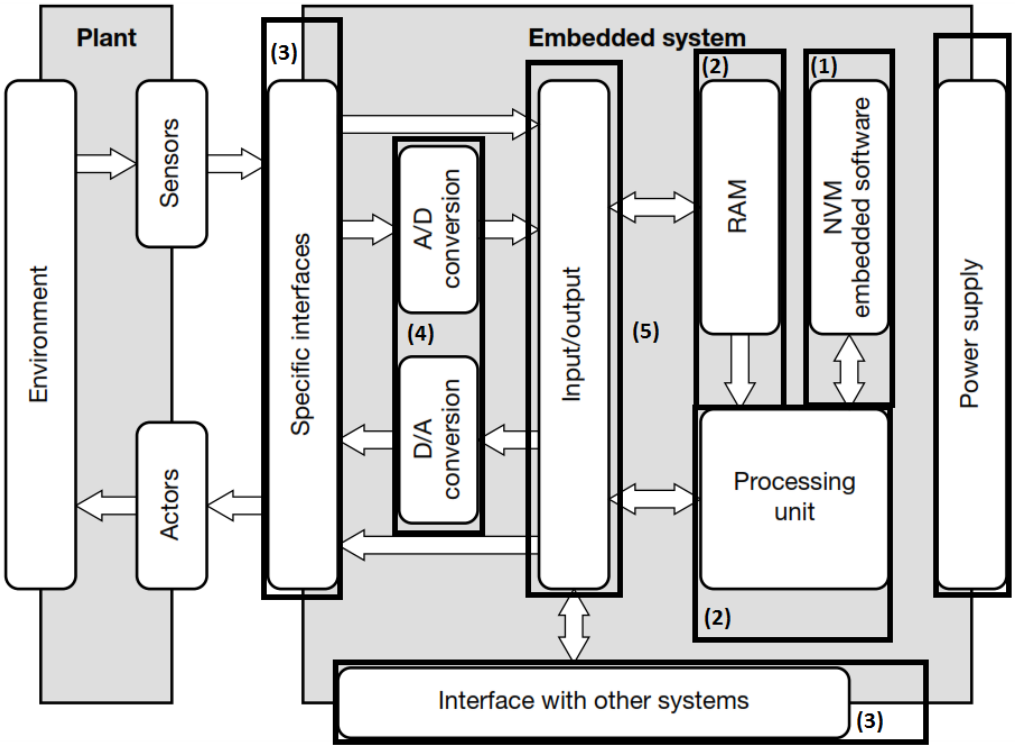


Figure 1: A generic scheme of an embedded system, sourced from [3,6].

Description of this diagram may be facilitated by following the numbered sections.

NVM (1) stores the embedded software for the system – system firmware. Frequently it is ROM, such as an EEPROM or flash memory. This software is compiled for the processing unit (2), which is the main functional unit in the embedded system, and it processes instructions and data, stored in RAM (2).

The processing unit (2) comes in several flavors, the most common being the microprocessor, a configuration containing only the processing unit and minimal I/O, and the microcontroller, which integrates all – or almost all the peripherals within a single package. It can be said that an embedded system is designed around its master processor [1, 130].

The data processed by the processor is acquired via a specific interface (3), which is connected to a dedicated I/O layer (5). As communication with the processor needs to be done digitally, the ADC (4) is utilized to convert the incoming signal from the sensor – and as many sensors work with analog data, a DAC (4) may be utilized by the system to communicate between the processor and the sensor.

2.2 Key Points to Designing Embedded System Software

Having established a generic model for an embedded system, it is desireable to introduce a few concepts that are relevant to producing the firmware for this project:

2.2.1 Program Flow

Determining a model of program flow is important when designing an embedded system. There exist three main models:

2.2.1.1 Polling

This is perhaps the simplest possible model. With polling, the processor is running the polling program in a loop and should a process require execution, execution is performed, and the loop is repeated.

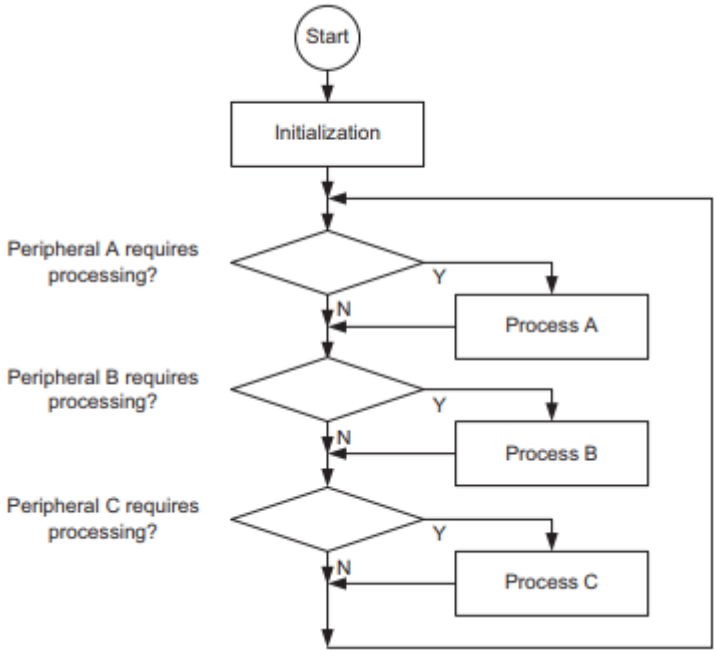


Figure 2: Polling Program Flow Model

This model works well for small and simple programs, but comes with several disadvantages: for example, if servicing one peripheral takes a long time, other peripherals will not get serviced during that period and responsiveness might suffer. Additionally, a

lot of power is wasted on empty loop cycles in which no peripheral is being addressed and the device is simply awaiting input. [4, 59]

2.2.1.2 Interrupt driven

An interrupt-driven model is somewhat more complicated, but greatly improves energy efficiency as compared to the polling model. In this model, the processor is asleep or running on low power until awoken by a pre-defined source (e.g peripheral interrupt), upon which it executes an interrupt service routine (ISR), services the peripheral, and goes back to low power mode once again. [4, 60]

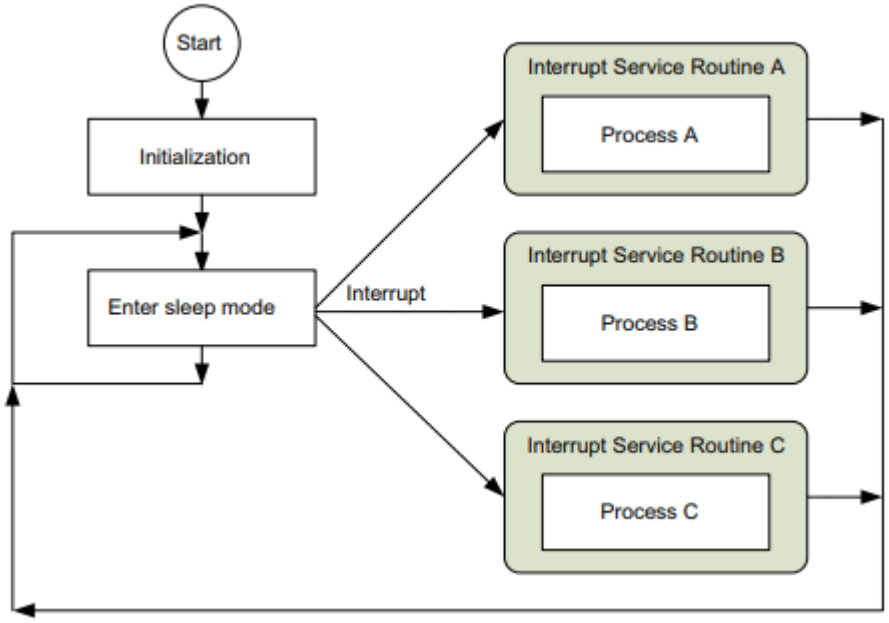


Figure 3: Interrupt driven program flow model

More careful planning is required, as interrupts need properly set priorities to guarantee that execution order is as required by the program: e.g, the ISR handling DMA transfer of ADC data cannot execute prior to the ISR handling ADC itself, as that might lead to incorrect data being transferred by DMA.

2.2.1.3 Combination of Polling and Interrupt Driven

There exist applications in which there is an advantage in a combination of the two methods above. One example would be an application in which emphasis is placed on quick execution of ISRs for the purpose of being able to serve more ISRs at once – allowing lower priority interrupts to be serviced faster.

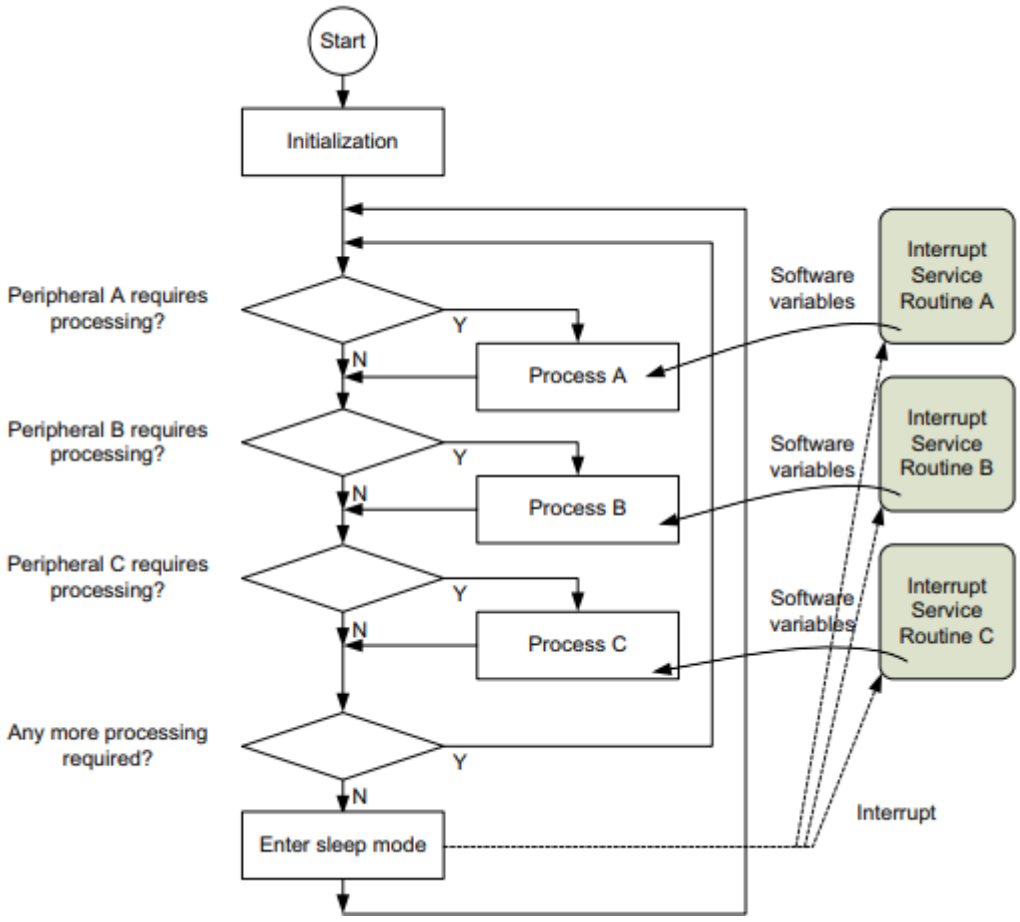


Figure 4: Mixed polling and interrupt-based program flow model

In this method, components of a given large ISR are divided into a smaller ISR, and a polling task. These two (or more) parts communicate with each other using software variables. Considering the nature of polling as compared to interrupts it can be said that while a task is not always able to be divided into parts easily, a large polling-based task does not stop peripheral interrupts from being serviced – and the benefits given by an interrupt-based approach are still present here, while no peripheral needs servicing, the system can run in low power mode. [4, 61]

2.2.2 Communication interfaces

2.2.2.1 DMA

DMA, or Direct Memory Access, is a programmable hardware module that exists in modern computer systems and allows hardware or peripherals access to memory directly, bypassing the CPU and letting it perform other tasks in the meantime. Memory to memory transfers are also supported. [8,1]

DMA is one of three primary data transfer methods, the other two being polling and the interrupt-based programmed I/O (PIO). To better illustrate the advantages of DMA, it is necessary to shortly present the two latter methods.

- In polling, the processor is dedicated to acquiring data, often by waiting for incoming data in a loop. It is a foreground process with direct CPU involvement.

- In PIO, interrupts regulate transfers, and as such, the processor is frequently interrupted from executing the main program while the interrupt is acquiring new data. A simplified presentation of that can be seen in figure 5:

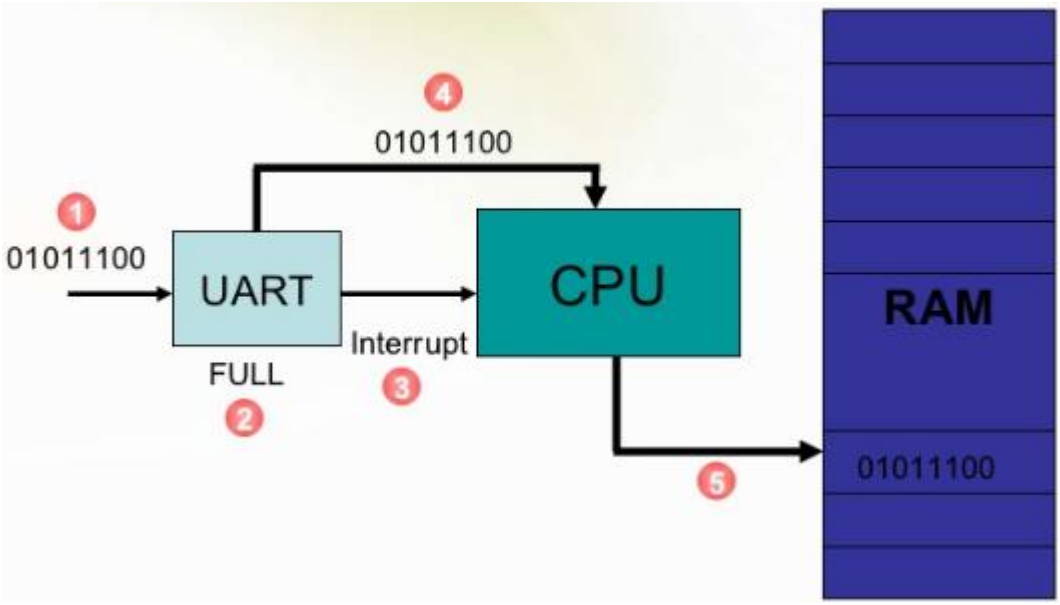


Figure 5: UART in PIO mode

The common denominator for PIO and polling is then the fact that both methods involve the CPU in a manner that is detrimental to overall system performance: program speed, latency, energy effectiveness are all affected parties - when optimal performance in any or all of those metrics is important, DMA is the right solution thanks to its nigh-complete CPU independence: Figure 6 presents UART implemented using DMA:

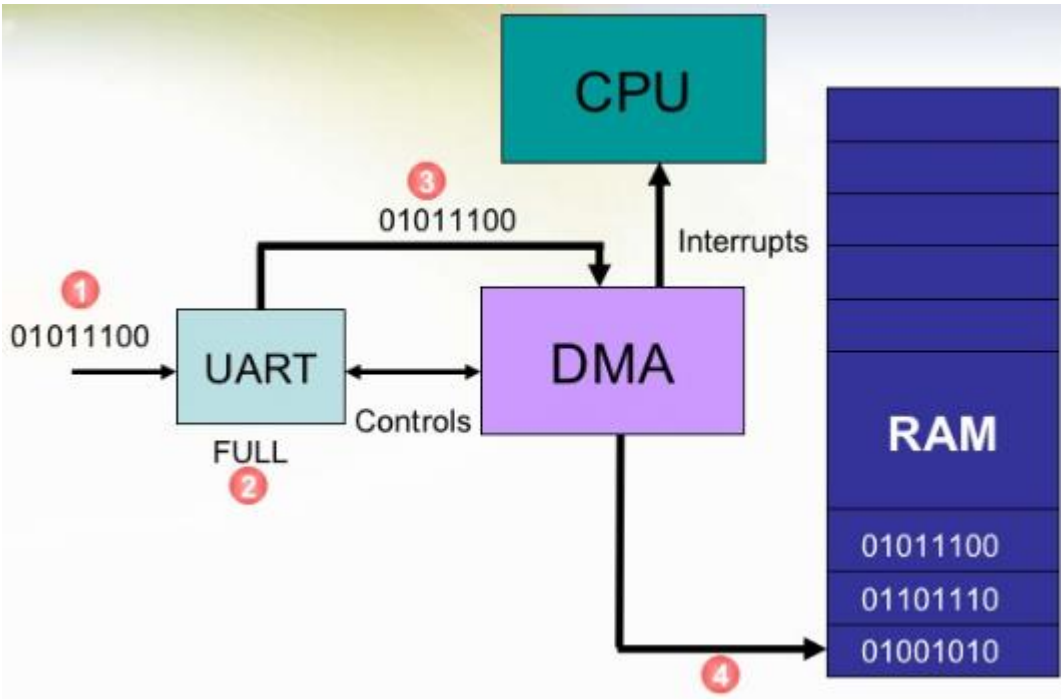


Figure 6: UART using DMA

2.2.2.2 USART

USART, short for Universal Synchronous and Asynchronous Transmitter is a communication protocol applying the principles of serial communication. It is one of the simplest communication protocols available with a minimum of two wires (one for transmission, the other for reception) required for implementation – upon which it is called UART, as there is no clock signal present to synchronize the transmission – synchronization is instead done using a fixed baud rate and bits that mark transmission start/end for every message.

The speed of a USART is a straightforward concept. It is measured in Bauds – a common unit in communications used to express bits per second of a given USART system. The time taken to transmit a single bit can be expressed with the following formula:

$$T_s = \frac{1}{f_s},$$

Where T_S stands for single bit duration, and f_S is the symbol rate.

A higher baud rate is hence capable of delivering the same amount of data faster, since the duration of a single bit is shorter. Additionally, oversampling is implemented to reduce chances of having errors in the transmission due to factors such as significant clock deviation.

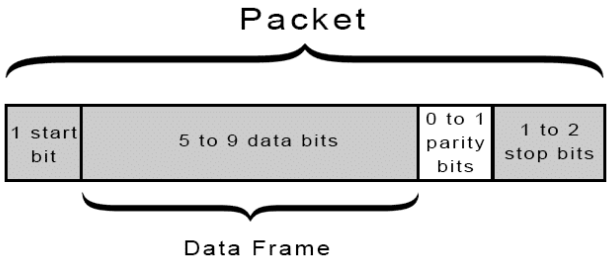


Figure 7: The configureable USART frame

As can be seen from figure 7, the sent frame is adjustable in length, and additionally one can choose to omit the parity bit in favor of more advanced error checking methods – as will be done in this project, opting instead for a CRC16 algorithm. A common configuration consists of 8 bits of data, one each start and stop bit, no parity bit, with a baud rate of 19200, and that will be the initial configuration used in this project.

2.2.3 Essential concepts

In this section a few concepts that are central to the project are introduced.

- Interrupt: An interrupt is an electrical signal (asynchronous or synchronous) asserted by a peripheral to the CPU upon which program execution is interrupted (hence the name), saved, and an interrupt service routine (ISR) – a block of code associated with the calling interrupt is executed [1,315]. Upon finishing the ISR, program execution continues from where it was halted.

An interrupt additionally possesses configurable priority levels so that in the case of multiple interrupt sources, the user is able to choose the priority of their assertion.

Lastly, it needs to be said that an interrupt service routine should be kept as short as possible: in systems lacking the feature of configurable interrupt priorities, should an unrelated interrupt trigger during the ISR, it might get processed with a noticeable delay leading to undesired system behavior.

- ADC: Analog to Digital Converter. A hardware device that reads an analog signal — typically a voltage — compares it to a reference voltage, and converts the resulting percentage to a digital value. The resolution of an ADC defines its precision in approximation of a given signal, and is defined as $\frac{\boldsymbol{V_{ref}}}{\boldsymbol{2^n}}$, where $\boldsymbol{V_{ref}}$ is the reference voltage, and $\boldsymbol{n}$ is the bit resolution.

  As a small example, consider a common 12-bit ADC. It offers 2^{12} = 4095 different voltage levels (digital values), with a resolution – difference between sequential levels of 3.3/4095 = ~805μv, assuming a $\boldsymbol{V_{ref}}$ of 3.3v.

  Lastly, it needs to be mentioned that any ADC must perform sampling at a frequency that is at least twice the frequency of the input – in other words, the Nyquist rate.

- DAC: Digital to Analog Converter. A hardware device that reads a digital signal and converts it to an analog output. It possesses many of the traits possessed by the ADC, such as resolution or sampling frequency. It may be said that it performs the inverse function to the ADC.

3 Implementation

3.1 Utilized hardware

A few factors have affected the hardware choice for this project – the chief factors being low power consumption and robustness. After some consultation with company representatives, the choices were narrowed down to the STM32L0x family from STMicroelectronics, and from there, the STM32L053R8T6 was chosen as the MCU for this project.

The STM32L053R8T6 is an ultra-low power platform, fully functional at voltages that are as low as 1.8V. It features a Cortex M0+ core at frequencies from 32kHz to 32Mhz, 64 KB of onboard flash memory, 8KB of SRAM, integrated USART with DMA functionality, and a generous amount of embedded I/O: there are a total of 51 GPIO pins present, out of which there are up to 16 12-bit ADC channels and two DAC output channels. [7,1]

As real hardware availability would not come for quite some time – it was due *sometime* in April, it was decided to use a development board to implement the initial prototype. The chosen board is from the budget-oriented STM32 Nucleo series, and is shown in figure 8 below:

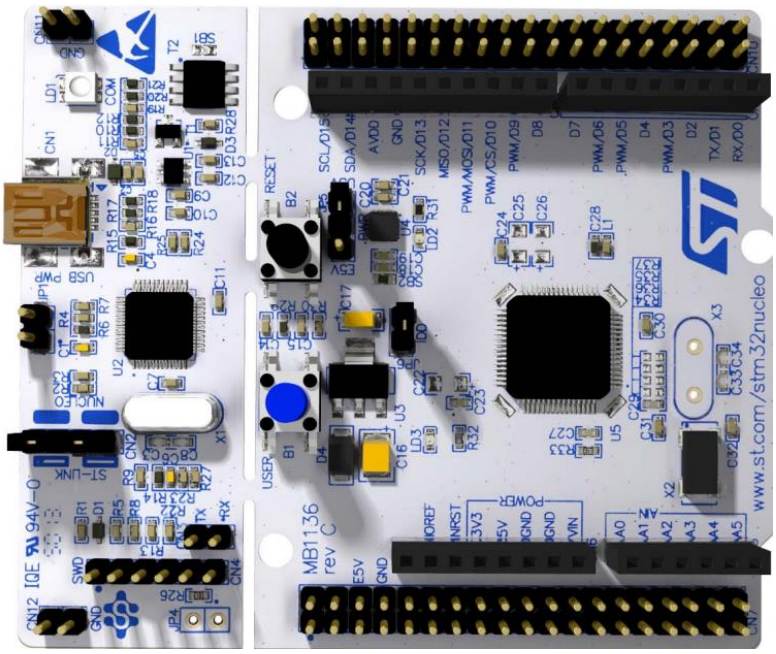


Figure 8: STM32L053R8T6 Nucleo development board

Every MCU needs a good IDE to program it, and for that purpose, the Keil µVision5 IDE, a part of the ARM-Keil MDK (Microcontroller Development Kit) has been chosen.

It is a full-featured IDE with a long legacy on ARM Cortex-M products with good integration of components that define the Cortex-M series of microcontrollers, such as CMSIS (Cortex Microcontroller Software Interface Standard, a software framework for embedded applications that run on Cortex-M based microcontrollers), extensive configurability options (both on the device level and in its user interface), many debug options, and last but not least, comprehensive documentation.

3.2 Programming Specifics

3.2.1 Working with Registers Directly

To facilitate understanding of most of the source code presented in this document, let us shortly present a few concepts that are *somewhat* uncommon in general-purpose programming, but necessary when working with registers directly – especially with embedded systems: those concepts are macros and bit manipulation.

Consider this example register, responsible for many features of a given USART:

29.8.2 Control register 2 (USARTx_CR2)

Address offset: 0x04

Reset value: 0x0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ADD[7:4]				ADD[3:0]				RTOEN	ABRMOD[1:0]		ABREN	MSBFIRST	DATAINV	TXINV	RXINV
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWAP	LINEN	STOP[1:0]		CLKEN	CPOL	CPHA	LBCL	Res.	LBDIE	LBDL	ADDM7	Res.	Res.	Res.	Res.
rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw				

Bits 22:21 **ABRMOD[1:0]**: Auto baud rate mode

These bits are set and cleared by software.

00: Measurement of the start bit is used to detect the baud rate.

01: Falling edge to falling edge measurement. (the received frame must start with a single bit = 1 -> Frame = Start10xxxxxx)

Bit 20 **ABREN**: Auto baud rate enable

This bit is set and cleared by software.

0: Auto baud rate detection is disabled.

1: Auto baud rate detection is enabled.

Figure 9: USART Control Register 2 and details of bits relevant to automatic baudrate detection, taken from [5, 804]

For the purpose of this example, our interest lies with the ABREN and ABRMOD bits, described in figure 9. They control the feature of auto-baud rate detection in a USART.

There are two ways to program each register:

1) Bit manipulation: This is perhaps the *quickest* way to program – directly writing the required feature bits to each register, as can be seen in listing 1:

```
USART2->CR2 |= (2UL << 21); //ABRMODE 01
USART2->CR2 |= (1UL << 20); //ABREN
```

Listing 1. Simple example for setting the required feature bits for auto-baudrate detection

In this case, we point at the required register, USARTx_CR2, and perform a bitwise OR operation (|=) on it: 1UL is an unsigned long data type (a 32-bit data type, fitting for this 32-bit register), shifted leftward 21 places – to bit 21, which controls ABRMOD. The register is clear at the time of system reset, and therefore the result of our bitwise OR operation is:

0 OR 1 = 1, which places a 1 in bit 21 of the USARTx_CR2 register and sets ABRMOD to 01 – falling edge to falling edge measurement. ABREN works in the same way.

Programming a register in this way is quick and efficient – however, unless the code is well documented and the reference manual Is followed while dissecting the code, understanding it might be difficult. Therefore, this method should be limited to simple functions and routines.

2) Using Macros: This is a more common and universal method than the above. Here is the same functionality implemented using macros:

```
USART2->CR2 |= USART_CR2_ABRMODE_1;
USART2->CR2 |= USART_CR2_ABREN;
```

Listing 2. Setting required feature bits using macro expansion

It can be asserted that listing 2 presents more understandable syntax – and the code is therefore more readable and easier to maintain. The *programmatic* implementation

however is still the same as shown in listing 1, which can be observed from the macro definitions shown in listing 3:

```
/*!< Auto Baud-Rate Enable*/
#define USART_CR2_ABRMODE_Pos          (21U)
#define USART_CR2_ABRMODE_1            (0x2U << USART_CR2_ABRMODE_Pos)
#define USART_CR2_ABREN_Pos            (20U)
#define USART_CR2_ABREN_Msk            (0x1U << USART_CR2_ABREN_Pos)
#define USART_CR2_ABREN                USART_CR2_ABREN_Msk
```

Listing 3. Macro definitions for the macros used in listing 2

The preprocessor macros define the bit positions in the register (ABRxx_Pos), and each macro also has a bit value to perform the setting operation with. Macro expansion takes care of the rest, and replaces every macro call with the value they define.

Lastly, it would be appropriate to summarize the main bitwise operations that may be used with the registers:

Table 1: Bitwise operations in C

Syntax	Meaning	foo, bar values	Example	Result	Use
~	Bitwise NOT	1	~foo	0	Invert bit
&	Bitwise AND	1,0	foo & bar	0	Clear bit
|	Bitwise OR	1,0	foo | bar	1	Set bit
^	Bitwise XOR	1,1	foo ^ bar	1	Toggle bit
<<	Bitwise left-shift	1,2	foo << bar	4	Shift toward MSB
>>	Bitwise right-shift	8,2	foo >> bar	2	Shift toward LSB

3.3 System Overview

An important step that needs to be undertaken prior to implementing features in code is reviewing the given project requirements thoroughly: there does not need to be more code than necessary, and a thorough review will save a significant amount of time, along with providing a base from which at least the very first prototype can be built.

3.3.1 System Requirements

The main objective of this project is to implement an add-in board for an existing sensor array. The add-in board is to serve as a user interface for the sensor array, and is

projected to exist in several variants with variable amounts of I/O available, with the following variant being the maximal configuration:

- 4xDigital input channels, 4xAnalog input channels.
- 6xDigital output channels, 1xAnalog output channel.
- Communication by USART, 19200 baud. Communication occurs every 25ms (40Hz refresh rate).
- Power supply for add-in is 1.9v rail from sensor array.

The I/O data from all the channels is to be stored on the sensor array and must be accessible on a r/w basis from a PC terminal, thereby updating I/O data on all devices involved. Figure 10 presents a schematic of the overall system:

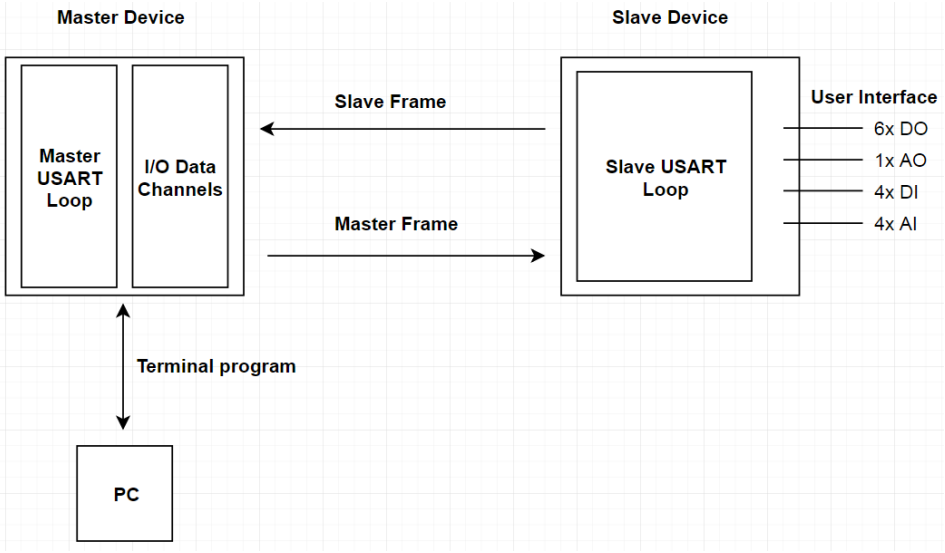


Figure 10: Generalized schematic for overall system

From this point on, the sensor array and add-in board will be referred to as the master device and the slave device, respectively.

3.3.2 Implementation guidelines

The given system is to be implemented in a configuration that is maximally robust and energy-effective: the power budget is very conservative, and, as a very generic guideline, assumes that *at least* 90% of the time the slave device CPU will be asleep, with its

wake time spent to process the given I/O. A general set of guidelines can be laid out based on those requirements:

- The system must be designed in a manner that facilitates implementation of a sleep/wake cycle: therefore, it must be designed around interrupts, ideally around the interrupt-driven model that is described in section 2.2.1.2.
- No hardware abstraction libraries are to be used in the firmware – opting for register-level programming instead: that is to avoid overhead from unnecessary "generic" code, thus saving CPU cycles and memory.
- DMA is to be used for all communications. DMA can transfer data bypassing the CPU, and thus it is instrumental in achieving a situation in which the CPU's *only task* consists of processing given input and output.

3.3.3 Initial Approach

The first step to implementing the system is picking a foundation to build upon: that foundation will be the master frame. The master frame is unconditionally sent by the master device at a rate of 25ms/40Hz, and thus it can be thought of as similar in functionality to a clock signal generated at a fixed rate and used for synchronizing actions. Hence, the foundation for synchronizing all the system actions is the 25ms cycle – that is the rate the master frame will be sent at, and that is additionally the rate at which the system will transfer channel data between master device and PC.

Considering the strict restrictions for power consumption mentioned in section 3.3.2, It is clear that there is a need for a mechanism that wakes the slave device upon reception of master frame: that mechanism is the __WFI(), or wait for interrupt instruction, which puts the MCU to sleep, and wakes it on occurrence of an interrupt. The interrupt source may be external – from a peripheral, for example. One such peripheral is DMA.

Investigating the reference manual further shows that the DMA peripheral offers an interrupt with several trigger flags, and of key interest is the "transfer complete" interrupt, which will trigger an interrupt upon transfer completion and wake the slave device, upon which data can be processed and a response frame can be built and sent by DMA back to the master device. The whole process is summarized by a flowchart, presented in figure 11:

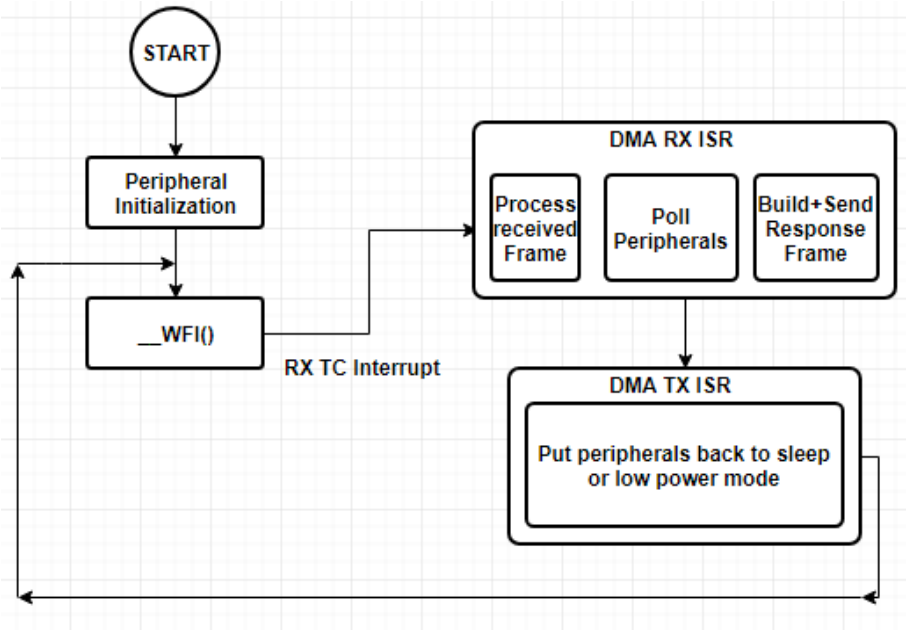


Figure 11: Projected/preliminary program flow, slave device.

It can be said that the sleep/wake process is unambiguous: provided there is no communication error on either side, the master frame will wake the slave from sleep at a rate of 40Hz/25ms. That timeframe includes the following actions, performed in a subsequent order:

1) Build and send output frame from master device.
2) Interpret output frame from master device.
3) Poll all peripherals, store all relevant data in buffers.
4) Build and send input frame to master device.

The length of steps 1 and 4 can be computed given the preliminary data: a baudrate of 19200bps, and two preliminary frame sizes for the input and the output frames. Formula (1) can be utilized to acquire frame duration:

$$F_s = \frac{1}{f_s} * (F_A * F_L) \quad (1)$$

Where F_s is frame duration in seconds, $\frac{1}{f_s}$ gives the time taken to transmit one bit, F_A is the amount of frames, and F_L is frame length.

Using formula (1), a single 8-bit frame with one start bit and one stop bit, without parity, at 19,2kbaud can be calculated to take:

F_S = 1/(19,2*10^3)*(1*10) = 0.0005208 = ~0.52ms.

Table 2 presents a summary of the projected I/O, its data types and their translations to UART frames, and the results of applying formula (1) to each frame:

Table 2: Master and slave I/O description and timing

Master Frame I/O	Type	UART	Bit time, 19.2kbaud
6xDO	1x uint8_t	1x uint8_t	0.52ms
1xAO	1x uint16_t	2x uint8_t	1.041ms
CRC16	1x uint16_t	2x uint8_t	1.041ms
Total	40	40	2.62ms
Slave Frame I/O	Type	UART	Bit time, 19.2kbaud
4xAI	4x uint16_t	8x uint8_t	4.166ms
4xDI	1x uint16_t	2x uint8_t	1.041ms
CRC16	1x uint16_t	2x uint8_t	1.041ms
Total	96	96	6.248ms

Steps (1) and (4) therefore take 2.62ms and 6.248ms, respectively. That gives the systems a wide time margin of 25-(2.62+6.248) = 16,132ms for performing steps (2) and (3) – therefore, it might be possible to halve the overall system baud rate to potentially save more power, while still having enough time to process all the necessary data: The total time taken by steps (1) and (4) with a baud rate of 9600 is calculated to be 17.736ms, which still gives a significant time margin of 7.264ms to steps (2) and (3).

3.4 Initializing hardware

3.4.1 System Clock

The center point from which all peripherals and hardware derive their clock speeds and some functionality is the system clock. From looking at datasheet [7, 22], There are a few sources for the system clock available on STM32L053: of interest are the HSI (High-Speed Internal) and MSI (Multi-Speed Internal) sources. An investigation into the low-power application note [6,12] yields the following specifications for said clocks:

Clock source	Use	Frequency	Consumption (typical)	Accuracy	Factory trimming	User trimmable
HSI16	Master clock	16 MHz	100 µA	0.4% typical(2)	Yes	Yes
MSI	Master clock	65.5 kHz 131 kHz 262 kHz 524 kHz 1.05 MHz 2.1 MHz 4.2 MHz	0.75 µA 1.0 µA 1.5 µA 2.5 µA 4.5 µA 8.0 µA 15 µA	0.5% typical(2)	Yes	Yes

Figure 12: System clock sources, STM32L053, taken from [6,12] and adjusted for relevance

From figure 12, one can observe a few key differences between these two master clocks: the first being peak clock speed, 4.2MHz for MSI vs 16MHz for HSI. The second being power consumption – at peak speeds, HSI consumes roughly seven times more current than MSI – and the ratio gets only larger as MSI speeds drop.

Considering that the application given in this project is both 1) requisitive of low power consumption and 2) does not require much processing power, it would be wise to build the system around the MSI clock source.

A quick look at the clock tree for the STM32L053 clock tree (appendix 1) suggests that the next step upon choosing a system clock is configuring the AHB (AMBA High-Performance Bus) which is the source for all clocks for the system. AHB configuration is done by setting a prescaler ratio – a mechanism that performs integer division on a high-speed clock signal, such as the system clock, thereby allowing the user to select a clock speed more fitting to the application. The AHB clock is directly connected to the CPU (HCLK), and is connected to system peripherals via APB – Advanced Peripheral Bus, which allows the peripheral clock to be scaled further using its own prescaler.

As can be seen from [5,179], the system utilizes MSI at 2.1MHz as the default clock source for the MCU at reset, and therefore, configuring MSI manually is unnecessary: additionally, for the sake of removing extra variables that could cause unexpected behavior in the system, AHB and APB will be initially prescaled by a ratio of 1 therefore, they will all operate at a speed of 2.1MHz. The prescalers are to be adjusted once the system has been tested to be stable. The function that implements this functionality is presented in listing 4.

```
void Configure_Peripheral_Clocks(void){
    RCC->CFGR |= RCC_CFGR_HPRE_DIV1;          // HCLK = SYSCLK, ~2.1MHz
    RCC->CFGR |= RCC_CFGR_PPRE1_DIV1;         // PCLK1 = HCLK
    RCC->CFGR |= RCC_CFGR_PPRE2_DIV1;         // PCLK2 = HCLK
}
```

Listing 4. Configuring CPU and peripheral clocks.

3.4.2 General process for initializing peripherals

Each peripheral that is to be used by the MCU requires an initialization process that may be generalized to a degree:

1. Enable clock to peripheral.
2. Clear any existing configuration bits on the peripheral.
3. (optional, but common) configure GPIO pin for peripheral.
4. Configure and/or calibrate the given peripheral.
5. Enable peripheral.

Listing 5 illustrates this process with an example:

3.4.3 Initializing Analog Input

```
void ADC_CALIBRATE(){
/*          RCC->APB2ENR |= RCC_APB2ENR_ADCEN; //enable ADC CLK

(1)         ADC->CCR |= ADC_CCR_LFMEN;       //low frequency ADC mode enable

*/          ADC1->CFGR2 |= ADC_CFGR2_CKMODE; //clk mode is PCLK/1, NECESSARY
                                             //for use with MSI CLK!

            /*if ADC enabled (ADEN), disable it (clear using ADDIS)
/*          this operation is necessary to give further commands*/
(2)         if ((ADC1->CR & ADC_CR_ADEN) != 0){
                ADC1->CR |= ADC_CR_ADDIS;
*/          }

(4)         ADC1->CR |= ADC_CR_ADCAL; //start calibration

            //wait until calibration (EOCAL) is over (bit=1);
            while ((ADC1->ISR & ADC_ISR_EOCAL) == 0){

            }
            ADC1->ISR |= ADC_ISR_EOCAL; //clear


}
```

```
void ADC_ENABLE(){


/*          ADC1->ISR |= ADC_ISR_ADRDY; //clear ADRDY by setting bit=1
(4)         //sampling time 160,5 ADC clk cycles
            ADC1->SMPR |= (ADC_SMPR_SMP_0 | ADC_SMPR_SMP_1 | ADC_SMPR_SMP_2);
*/
(5)         ADC1->CR |= ADC_CR_ADEN; //enable ADC
            //wait until ADC is ready
            if ((ADC1->CFGR1 & ADC_CFGR1_AUTOFF) == 0){
                while ((ADC1->ISR & ADC_ISR_ADRDY) == 0){
                }
            }
}
```

Listing 5. Initializing ADC.

The ADC module will serve the 4 pins that offer analog input on the slave device.

3.4.4 Initializing Digital Inputs

Digital inputs require configuring a set of relevant pins in input mode. Traditionally, a digital input works in pull-up mode – that is, a logical 0 signifies user input, while a logical 1 is present otherwise. The four digital inputs are configured according to the following template:

```
void Configure_Pushbuttons(void){
    RCC->IOPENR |= RCC_IOPENR_GPIOCEN;           //GPIOC CLK EN
    //PC13 = JB DI1 = B1 on evaluation board
    GPIOC->MODER   &= ~GPIO_MODER_MODE13;        //select PC13 in input mode
    GPIOC->PUPDR   |= GPIO_PUPDR_PUPD13_0;       //pull-up mode
    GPIOC->OSPEEDR |= GPIO_OSPEEDER_OSPEED13_0;  //medium sampling speed, 2MHz
}
```

Listing 6. Template for configuring a digital input on the slave device

To shrink the output frame, all slave digital inputs can be packed into one variable – a bitmask, and then interpreted on the receiving side: the packing is performed by the following function.

```
uint8_t Check_Button_State(void){
    uint8_t buttonState = 0;
    buttonState |= !(GPIOC->IDR & (1 << 13)) << 0;
    buttonState |= !(GPIOA->IDR & (1 << 8))  << 1;
    buttonState |= !(GPIOC->IDR & (1 << 7))  << 2;
    buttonState |= !(GPIOA->IDR & (1 << 5))  << 3;
    return buttonState;
}
```

Listing 7. Packing multiple digital inputs into a bitmask.

The master checks the set bits in the frame it receives, and thus is made aware of what inputs have been toggled on the slave device, as it is aware of exactly what input each set bit signifies. The process is very similar to the packing function in listing 7, it is not detailed here.

3.4.5 Initializing Analog Output

Analog output is performed by the DAC module. Configuration of the DAC requires enabling the peripheral, with manually configuring the DAC output pin as an analog pin a highly desirable action, as it prevents parasitic power consumption on the pin.

```
#define MAX_VALUE   0xFFF

void DAC_INIT(void){
    RCC->IOPENR  |= RCC_IOPENR_GPIOAEN;  //Enable the peripheral clk of GPIOA
    GPIOA->MODER |= GPIO_MODER_MODE4;    //Select analog mode for PA4
    RCC->APB1ENR |= RCC_APB1ENR_DACEN;   //Enable the peripheral clk of DAC
    DAC->CR = DAC_CR_BOFF1 | DAC_CR_EN1; //Enable the DAC ch1
}

void DAC_WRITE(uint16_t value){
    if (value > MAX_VALUE){
        DAC->DHR12R1 = MAX_VALUE;
    }
    else{
        DAC->DHR12R1 = value;
    }
}
```

Listing 8. Initializing and using the DAC

Reading the value from the DAC is performed by simply reading the register.

```
DAC_Value = (uint16_t)(DAC->DHR12R1);
```

3.4.6 Configuring USART and DMA

3.4.6.1 USART

USART, first and foremost, requires the pins it is connected to be reconfigured from their standard GPIO functionality to an alternate function mode – a blanket term for a pin mode required by many peripherals whose functionality extends beyond reading/writing registers in GPIO mode. AF is required also by I²C and SPI, among others.

```
void Configure_GPIO_USART1(void){
  RCC->IOPENR |= RCC_IOPENR_GPIOAEN;  //Enable GPIOA clock

  /* Select AF mode (10) on PA9 and PA10 */
  /* AF4 for USART1 signals */
  GPIOA->MODER = (GPIOA->MODER & ~(GPIO_MODER_MODE9|GPIO_MODER_MODE10))\
                 | (GPIO_MODER_MODE9_1 | GPIO_MODER_MODE10_1);
  GPIOA->AFR[1] = (GPIOA->AFR[1] &~ (0x00000FF0))\
                  | (4 << (1 * 4)) | (4 << (2 * 4));
}
```

Listing 9. Initializing alternate functionality for GPIO pins 9 and 10

Beyond that, configuring USART is a process that involves setting its parameters to reflect the expected frame format and features in this project: 8 data bits, 1 each start and stop bit, no parity is the default configuration, and therefore no programming is required to set those parameters. The code comments describe the rest of the settings.

```
void Configure_USART1(void){
  RCC->APB2ENR |= RCC_APB2ENR_USART1EN; //Enable USART#1 clock

  // PCLK/BAUD, oversampling by 16, 19200 baud
  USART1->BRR = 2048000 / 19200;

  //Auto-BaudRate Detection based on start bit and first data bit
  USART1->CR2 = USART_CR2_ABRMODE_1;
  USART1->CR2 = USART_CR2_ABREN;

  //Enable DMA mode in transmit and receive
  USART1->CR3 = USART_CR3_DMAT | USART_CR3_DMAR;

  //USART enabled for transmission and reception
  USART1->CR1 = USART_CR1_TE | USART_CR1_RE | USART_CR1_UE;

  while((USART1->ISR & USART_ISR_TC) != USART_ISR_TC){
  }
  USART1->ICR = USART_ICR_TCCF; //clear Transmission Complete flag
}
```

Listing 10. Initializing USART1 on slave device

3.4.6.2 DMA

DMA requires special care in its configuration: its entire functionality, as said earlier, consists of offloading memory transfers from the CPU – and since the sources and destinations of memory transfers are many (be it peripheral to memory, vice versa, or memory to memory), each given DMA channel needs to be specifically configured to its respective task: The following text follows the order of actions performed in listing 11 and describes each step in additional detail.

The first step past enabling the peripheral clock is tying the DMA peripheral to a relevant data register – source memory: in this case, the registers are the USART transmit and receive registers. The next step involves tying the DMA a buffer – destination memory. Arrays in the relevant data type are defined, and DMA is given their addresses and sizes. The steps listed so far are generic, and recur in most DMA applications.

The last step consists of enabling and configuring specific DMA features. In the case of this application, the DMA configuration needs to match USART frame parameters (specifically, 8-bit length), and the required interrupt type: an interrupt is to be called on completing frame reception and completing frame transmission.

```
void Configure_DMA1(void){
  RCC->AHBENR |= RCC_AHBENR_DMA1EN; //enable periph.clk for DMA1

/**Enabling DMA for transmission
* DMA1, Channel 2 mapped for USART1TX
* USART1 TDR for peripheral address
* stringtosend for data address
* Memory increment, memory to peripheral | 8-bit transfer | transfer complete
interrupt**/
  DMA1_CSELR->CSELR = (DMA1_CSELR->CSELR & ~DMA_CSELR_C2S) | (3 << (1*4));
  DMA1_Channel2->CPAR = (uint32_t)&(USART1->TDR);
  DMA1_Channel2->CMAR = (uint32_t)stringtosend;
  DMA1_Channel2->CCR = DMA_CCR_MINC | DMA_CCR_DIR | DMA_CCR_TCIE;

/**Enabling DMA for reception
* DMA1, Channel 3 mapped for USART1RX
* USART1 RDR for peripheral address
* stringtoreceive for data address
* Data size given
* Memory increment, peripheral to memory | 8-bit transfer | transfer complete
interrupt**/
  DMA1_CSELR->CSELR = (DMA1_CSELR->CSELR & ~DMA_CSELR_C3S) | (3 << (2 * 4));
  DMA1_Channel3->CPAR = (uint32_t)&(USART1->RDR);
  DMA1_Channel3->CMAR = (uint32_t)stringtoreceive;
  DMA1_Channel3->CNDTR = sizeof(stringtoreceive);
  DMA1_Channel3->CCR = DMA_CCR_MINC | DMA_CCR_TCIE | DMA_CCR_EN;


  NVIC_SetPriority(DMA1_Channel2_3_IRQn, 0); //NVIC enabled, max priority, DMA
channels 2-3
  NVIC_EnableIRQ(DMA1_Channel2_3_IRQn);
}
```

Listing 11. Configuring DMA for transmission and reception

The processes of transmission and reception in the slave device are to be done within an interrupt handler, defined in listing 12.

```
void DMA1_Channel2_3_IRQHandler(void){
  if((DMA1->ISR & DMA_ISR_TCIF2) == DMA_ISR_TCIF2)
  {
    DMA1->IFCR = DMA_IFCR_CTCIF2; //clear TC flag
    LPSLEEP();
    responseState = 1; //transfer done
  }
  else if((DMA1->ISR & DMA_ISR_TCIF3) == DMA_ISR_TCIF3)
  {
    /*process received frame*/
    DMA1_Channel3->CCR &=~ DMA_CCR_EN;
    DMA1_Channel3->CNDTR = sizeof(stringtoreceive2);
    DMA1_Channel3->CCR |= DMA_CCR_EN;

    DMA1->IFCR = DMA_IFCR_CTCIF3; //clear TC flag

    responseState = 0; //response in progress
    Acquire_JB_Input_Data();
    Build_JB_Input_Frame();

    /*send response frame*/
    DMA1_Channel2->CCR &= ~DMA_CCR_EN;
    DMA1_Channel2->CNDTR = sizeof(JB_Input_Frame);
    DMA1_Channel2->CCR |= DMA_CCR_EN;
  }
}
```

Listing 12. DMA interrupt handler

This interrupt handler in listing 12 can be viewed as a rudimentary state machine that consists of two states: transfer in progress (=0) and transfer done (=1), as denoted by the responseState variable. It controls program flow for the slave device:

- Reception of a frame triggers a device wake, data acquisition, and assembly of a response frame. responseState becomes 0 and the transfer is denoted in progress.
- Upon assembly of the frame, it is transmitted to the master device. That denotes that the transfer as done, responseState = 1, and the device goes back to sleep.

The sleep process is implemented using the aforementioned __WFI() function, which is placed in a while loop in the main function. Therefore, it can be said that at any point outside of the interrupt handler, the system is asleep, waiting for an interrupt to happen.

3.5 Configuring Low-Power Features

A set of regulations for entering low power sleep mode can be found from [5,157]. They are applicable outside of sleep mode as well, and can be summarized like so:

1) Switch off the flash memory.
2) Disable clocks to unnecessary peripherals.
3) Decrease system clock speed.
4) Set voltage regulator to low-power mode.
5) Enter sleep mode via any entry point to it: in this case, __WFI().

There is no single function or register that performs all the above tasks given by STM or ARM, so one must instead implement the regulations step-by-step on the register level. Steps 2,3,5 are implemented outside of the function below., e.g in listings 4 (step 3) and 13 (steps 1 and 4).

```
void Configure_LowPowerMode(void){
    FLASH->ACR |= FLASH_ACR_SLEEP_PD; //Power down mode for flash
    PWR->CR |= PWR_CR_LPSDSR;         //Voltage regulator in low power mode
    PWR->CR |= PWR_CR_LPRUN;          //Low power run
}
```

Listing 13.Configuring Low Power Mode for the MCU

3.6 Building and Parsing USART Frames

With the program flow firmly established and all peripherals initialized, it is time to implement the main functionality of the system: algorithms that process a received frame, and construct a response frame.

With the focus of this project being on the slave device, the set of algorithms processing the received frame and send a response frame were built first. Much trivial but necessary code that acquires data is omitted from this section to keep this section short and to the point.

When building a frame that consists of data that is wider than eight bits, for example, ADC measurement data – the ADC precision is 12-bit, and its data is contained in a uint16_t variable – too wide for one USART frame, one must break that data up into eight-bit chunks. That is done by shifting the bits rightward by x*8 places, as such:

```
uint16_t example = 0x19B4; //Decimal = 6580, Binary = 00011001 10110100
uint8_t  newLSB  = example >> 0; //Binary = 10110100
uint8_t  newMSB  = example >> 8; //Binary = 00011001
```

On the receiving side, the inverse is necessary, and the received numbers are shifted leftward by the same x*8 places, thereby placing them in the correct place according to bit order.

```
uint16_t exampleRebuilt = 0;
exampleRebuilt = (newLSB | newMSB << 8); //00011001 10110100
```

Listing 14.Example of bitwise shifts to split and rebuild 16-bit integer.

After applying those rules with some automation, the resulting algorithms for constructing and parsing the slave output frame are:

```
void Build_JB_Input_Frame(){
    uint8_t idx_a = 0, idx_b = 0;
    uint16_t CRC_Sum = 0;

    /*Compound the ADC inputs*/
    for (idx_a = 0; idx_a < (sizeof(adc_Results)/sizeof(adc_Results[0]));
++idx_a){
        JB_Input_Frame[idx_b] = adc_Results[idx_a] >> 0;
        JB_Input_Frame[idx_b+1] = adc_Results[idx_a] >> 8;
        idx_b += 2;
    }
//at end of analog inputs, digital states mask
    JB_Input_Frame[idx_b] = GPIO_States;
    ++idx_b;

    CRC_Sum = crc16(JB_Input_Frame, sizeof(JB_Input_Frame)/sizeof(JB_In-
put_Frame[0]), 0xFFFF);
    JB_Input_Frame[idx_b]   = CRC_Sum >> 0; //LSB
    JB_Input_Frame[idx_b+1] = CRC_Sum >> 8; //MSB

}
```

Listing 15.Decomposing given data into 8-bit chunks for USART.

```
void Rebuild_JB_Frame(){
    uint8_t idx_a = 0, idx_b = 0;

    uint16_t JB_Input_LSB = 0, JB_Input_MSB = 0;

    for(idx_b = 0; idx_b < JBInFrameLength; ++idx_b){

        JB_Input_LSB = JB_Input_Frame[idx_b];
        JB_Input_MSB = JB_Input_Frame[idx_b+1] << 8;

//Every two idx_b ticks, we have one uint16_t value to build with index idx_a
        if(idx_b % 2 == 0){
            rebuiltframe[idx_a] = JB_Input_LSB+JB_Input_MSB;
            idx_a++;
        }
    }
}
```

Listing 16. Rebuilding received 8-bit data chunks back into original form.

It is easy to see the concepts introduced in the shifting example applied in listings 15 (right shift) and 16 (left shift). These algorithms additionally attempt to make the process of building and parsing the frame somewhat generic, and not reliant upon hard-coded loop bounds – by using the sizeof() function along with some preprocessor definitions that handle frame length (e.g JBInFrameLength), adjusting the algorithms to a handle a different configuration of slave device (e.g different amount of I/O) is a matter of a simple decision structure.

The process of frame construction and parsing is similar enough on the master device side that code for it does not need repeating.

3.7 Main Program

The main program in this interrupt-driven case corresponds the outline laid out in section 3.3.3: upon peripheral initialization, the system is sleeping, waiting for an interrupt in a loop – and upon reception, services the interrupt, and goes back to sleep.

```
int main(){
    Configure_Peripheral_Clocks();
    Configure_GPIO_USART1();
    Configure_DMA1();
    Configure_USART1();
    Configure_Pushbuttons();
//  Configure_USART2();     /*USART2          */
//  ENABLE_USART2_DMA();      /*For PC ST-LINK */
    INIT_JB_HW();
    LPSLEEP();

    SCB->SCR |= (1UL << 1); //sleep on exit feature enabled

    while(1){
        __WFI();
    }
}
```

Listing 17. Main program, slave device

4 Tests and Optimizations

4.1 Test Setup

The test setup consists of:

- The slave device, prototyped on the development board. The ADC inputs are fed by the same voltage to simply the setup, the digital input is onboard button B1.
- A breadboard, which facilitates wire organization and signal measurement. It is employed as a sort of a bridge between the master, slave, and oscilloscope.
- The master device. Only two wires coming from it are relevant, USART TX and RX.
- A PicoScope 2205 Oscilloscope. Required to determine timing of USART frames.
- A Keithley 2000 multimeter to perform current consumption measurements.

Appendix 2 visualizes the test setup with pictures.

4.2 Testing USART

The initial set of USART tests was performed without implementing error handling code. The interest was to get communications working as fast as possible - and that proved to be a major oversight, resulting in permanent desynchronization between the master and slave. The issue easily appears if the slave is started while the master is already in the middle of transmitting a frame: that leads to the slave receiving part of an old frame and part of a new one, filling the buffer at the wrong data point, triggering a reply interrupt out of sync.

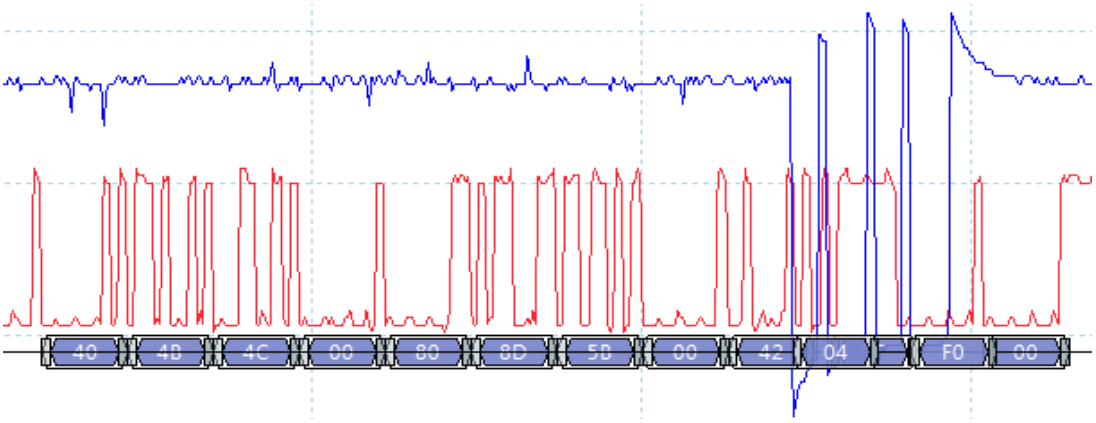


Figure 13: Oscilloscope data, USART timing mismatch, initial testing

As can be seen from figure 13, the frame timing is mismatched: the blue frame (response from slave) overlays the red frame (master frame), indicating a buffer fill at the wrong time, since the receive buffer full interrupt handles sending the response frame from the slave.

The solution to this problem is implementing a CRC-based error checking mechanism to control program flow at the response stage. That can be said when considering the following: The position of the CRC bits in the frame is known ahead of time - and a position mismatch, such as in this case, will make the fact that there is an error obvious. Additionally, the CRC algorithm is sensitive to the smallest differences, as small as an additional 0 value in the data sample. Therefore, a frame received out of order will inevitably result in the receiver calculating a CRC that does not match the CRC it received from the master.

Upon noting a CRC mismatch, the transmission line is disabled, and the received frame is discarded. No response frame is built/sent. The slave takes account of the point on which the RX DMA line goes idle, disables the reception line, and waits an amount of bit times that is equivalent to the length of one whole master frame – to surely skip any more out-of-place data. When the wait is over, an idle line interrupt is triggered, and enables reception and transmission. The system is re-synchronized by the next master frame it receives. The following schematic visualizes this procedure:

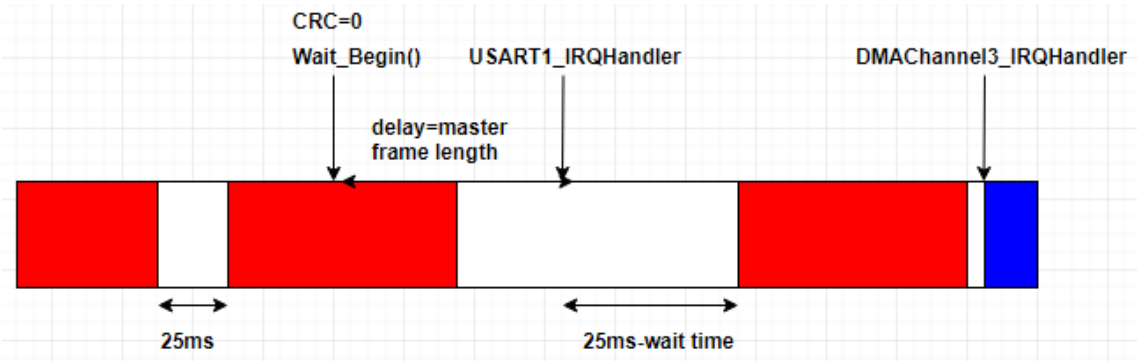


Figure 14: Frame Resynchronization Procedure

Listing 18 on the next page implements this method. The code from this listing is merged into a modified version of the DMA interrupt handler, shown earlier in listing 12.

```
    CRC = Compare_Received_CRC();

    if(CRC == 1){
        /*build ONLY if CRC correct*/
        Build_JB_Input_Frame();
        DMA1_Channel2->CCR &= ~DMA_CCR_EN;
        DMA1_Channel2->CNDTR = sizeof(JB_Input_Frame);
        DMA1_Channel2->CCR |= DMA_CCR_EN;
    }
    else if (CRC==0){
        delay = 50; //delay of 5 sets of 10 bits as in the master outframe

        USART1->CR2 = USART_CR2_RTOEN;  //enable receiver idle line interrupt
        USART1->RTOR |= (delay << 0);
        DMA1_Channel3->CCR &= ~DMA_CCR_EN;  //disable DMA reception

        //clear received frame from data
        memset_volatile(NDX_Output_Frame,0,5);

        /*Reset data size, address to receive*/
        DMA1_Channel3->CMAR = (uint32_t)NDX_Output_Frame;
        DMA1_Channel3->CNDTR = sizeof(NDX_Output_Frame);
        DMA1_Channel3->CCR |= DMA_CCR_EN;  //channel enable
    }
  }
}

void USART1_IRQHandler(void){
    DMA1_Channel2->CCR &= ~DMA_CCR_EN;  //disable DMA transmission
    USART1->RTOR &= ~(1UL << 0);        //reset bit wait value
    USART1->CR2 &= ~USART_CR2_RTOEN;    //disable receiver line idle interrupt
    USART1->ICR = USART_ICR_RTOCF;      //clear USART RTOF Interrupt Flag
}
```

Listing 18. Extension of DMA interrupt handler with CRC-based program flow

To maximally strain the system, every other test frame sent was sent with a wrong CRC of the value 0xAAAA. The correct frame was constant to isolate possible test variables, and its CRC was 0xFD42. The results can be seen in figures 15 and 16:

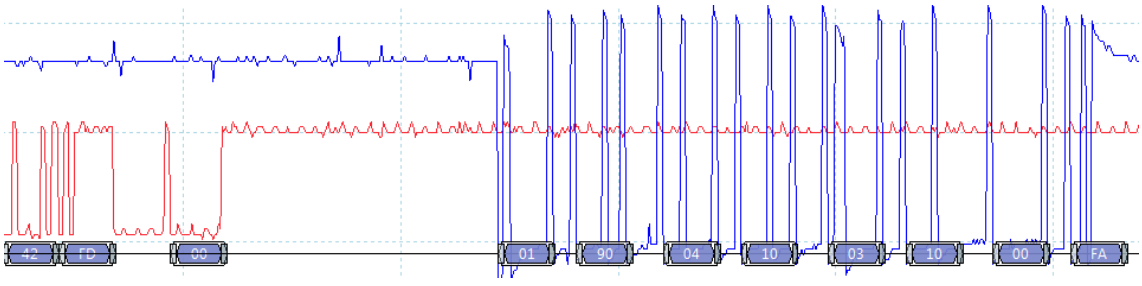


Figure 15: Response to correct CRC

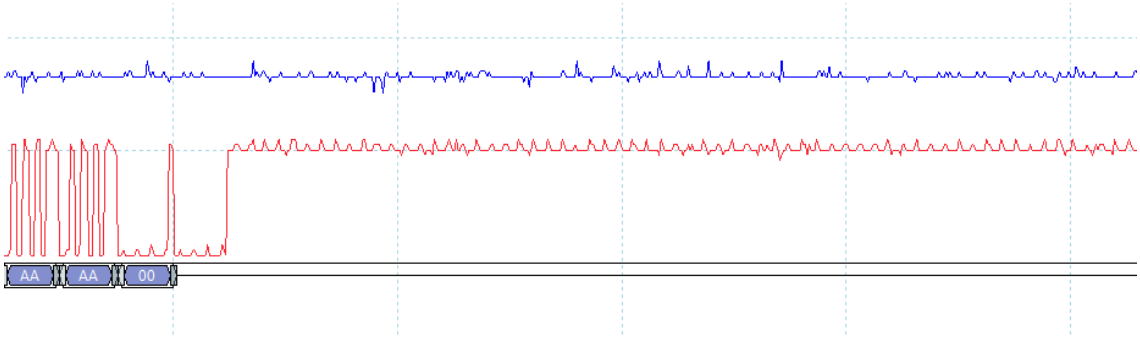


Figure 16: Line is silenced in response to wrong CRC

With transmission working reliably in either direction, some measurements on frame length were performed, shown in table 3. Each case easily corresponds to the required 25ms timeframe for all processes, and therefore, the last one (9600, PCLK/4) is picked for the project: it consumes the least current, as will be shown in the next section.

Table 3 :Frame transmission and processing times at differing baudrates and peripheral clocks

Speed	Master Frame (ms)	Slave Frame (ms)	Wake Time (ms)	Total (ms)
19200 baud, PCLK/1	2.71	6.3	1.11	10.12
19200 baud, PCLK/2	2.75	6.41	1.95	11.11
9600 baud, PCLK/1	5.27	12.4	1.34	19.01
9600 baud, PCLK/4	5.34	12.34	3.5	21.18

4.3 Measuring and Optimizing Power Consumption

Measuring MCU current consumption is facilitated by a dedicated pin available on the board for that exact purpose - IDD. Upon removing the relevant jumper, the MCU voltage source switches down to a 3.3v supply, and current may be measured like so:

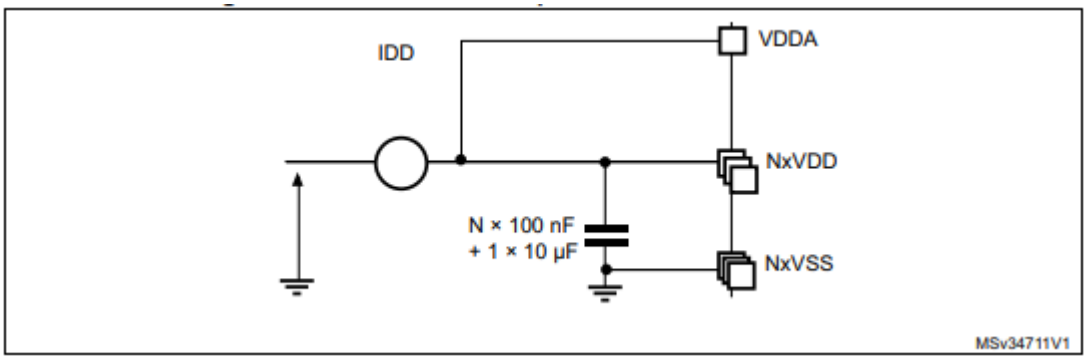


Figure 17: Current consumption measurement scheme, STM32L053xx MCU, taken from [7,52]

For each set of measurements, a baseline measurement that corresponds to the highest possible power consumption has been established – that measurement is performed without any power optimizations and without putting the device to sleep via __WFI.

As the transmission rate of the master device is rather high at 40Hz, sudden changes in current during slave response may be difficult to observe using an ammeter: therefore, all measurements have been performed in *moving average* mode. Table 4 presents the measurement results:

Table 4: MCU current consumption measurement results, ran from ROM unless otherwise noted.

Configuration	AVG Measured current, µA	% of baseline
PCLKDIV = 1, BAUD=19,2K, no __WFI(),	579	100
PCLKDIV = 1, BAUD=19,2K, no __WFI(), V_reg min	559	96.54576857
PCLKDIV = 1, BAUD=19,2K, no __WFI(), Run from RAM	550	94.99136442
PCLKDIV = 2, BAUD=19,2K, no __WFI()	450	77.72020725
PCLKDIV = 1, BAUD=19,2K, __WFI()	402	69.43005181
PCLKDIV = 1, BAUD=19,2K, __WFI(), OE	402	69.43005181
PCLKDIV = 2, BAUD=19,2K, __WFI(), OE, V_Reg min, run from RAM	357	61.65803109
PCLKDIV = 1, BAUD=9,6K, no __WFI()	574	100
PCLKDIV = 1, BAUD=9,6K, no __WFI(), V_reg min	555	96.68989547
PCLKDIV = 1, BAUD=9,6K, no __WFI(), Run from RAM	540	94.07665505
PCLKDIV = 4, BAUD=9,6K, no __WFI()	365	63.58885017
PCLKDIV = 1, BAUD=9,6K, __WFI()	390	67.94425087
PCLKDIV = 1, BAUD=9,6K, __WFI(), OE	390	67.94425087
PCLKDIV = 4, BAUD=9,6K, __WFI(), OE, V_Reg min, run from RAM	312	54.3554007

It needs to be noted that PCLK runs at HCLK frequency – the peripheral bus frequency is the same as the frequency of the CPU. OE stands for “Optimization Enabled”, and the optimizations made correspond to the ones described section 4.4.

Each variable was tested separately to isolate its effects from the rest of the possible optimizations. Some notable observations from table 4 are:

- Power savings achieved by setting the voltage regulator to low power mode and running the program from RAM (=flash ROM is off) are consistent regardless of baud rate, albeit small: an average of 3.5% and 5.5%, respectively.
- Power savings from reducing the clock speed supplied to the peripherals and CPU are especially significant: a 23% reduction in power consumption for 19.2Kbaud, and a 37% reduction for 9,6Kbaud. The large gap between the two results can be attributed to the fact that peripheral clock runs at the same speed

as the CPU clock, and therefore, at 9,6Kbaud, the CPU clock is twice lower than at 19,2Kbaud.

- Compiler-based optimization appears useless in a simple program such as this, offering no measurable difference in power consumption.
- With either baud rate, __WFI reduces current consumption by a significant 31-33%. The small disparity is likely due to the different time periods that were taken to measure each case, leading to different average currents measured.

At this rather early point in project development there is no exact power consumption requirement that the slave device must conform to – just a very general guideline that suggests that the CPU must be kept asleep at least 90% of the time, which, ideally should be a major contributing factor to the overall low power consumption of the device. However, as the results in table 4 show, the configuration that saves the most power is 9600 baud, PCLK/4, with a 45.645% reduction in power consumption from the baseline. Table 3 shows that that configuration takes 3.5ms to process a received frame and build a response, and therefore spends 14% of the total 25ms timeframe with its CPU running. It can be therefore said that putting the device to sleep for a more prolonged time is not necessarily the way that saves the most overall power.

4.4 Compiler-side optimizations

Optimizations that are done using the compiler are the easiest to apply. This project is built on ARMv6 compiler, and in this case, applying optimizations does not require revising the existing code: it is enough to set a few flags and tick a few boxes in the compiler/linker.

There are official recommendations for optimization offered by ARM via application note 202, and this project follows the configuration recommended for optimizing for best performance, as described in [9,7].

There are three procedures that optimize the firmware for best performance.

1) Cross-module optimization, also known as linker feedback, takes information from a prior build and uses it to remove unused functions from the next build, thus

saving memory by reducing the overall code size. Modules are also able to share inline code, thus improving overall performance. [9,3]

2) Optimization level flag -O3 applies code optimization that is heavily in favor of code performance at the expense of debug functionality and code size. [9,3].

3) Optimization flag -Otime instructs the compiler to optimize the code for the fastest execution time, with a possible tradeoff of a larger code size. [9,4].

The code was first compiled with only cross-module optimization enabled so that the extent of code size reduction can be observed – the original code took 2774kb, and cross compilation reduced it to 1884kb. Considering that the code size reduction was significant, code size increase via optimizations -O3 and -Otime should not present a problem in comparison. Therefore, they will be enabled to acquire additional performance improvements.

4.5 Running firmware entirely in RAM

The small resulting code size, optimized or otherwise, suggests that it is possible to run the entirety of the firmware in RAM, thus allowing flash memory to be turned off for the entirety of the duration that the system is running. While the datasheet projected reduction in power consumption is around 12.3μA, the actual reduction was instead measured to be around 30μA.

Directing the program to run entirely in RAM is as simple as pointing the IDE to load code to the memory addresses for RAM (0x20xx) instead of the addresses for flash (0x80xx).

The memory map file can then be checked to verify that the procedure has been done correctly: the presence of user-made made functions in locations beginning with 0x20 is indicative of the process being done correctly.

5 Conclusions

The primary goal of this project was to implement firmware for an add-in board to an existing sensor array – a master device. The add-in board was to perform I/O functionality for the sensor array via several digital and analog inputs and outputs, transmitted by USART. Special emphasis was placed on reducing the power consumption of the add-in board and making the communication link additionally robust.

Extensive testing confirmed that the firmware fulfills its intended purpose well: communication is ideally synchronized to the master device and easily responds within the required time frame, with time to spare. The data communicated from either master side or slave side is parsed correctly, and should an error occur at any point, the system *will* automatically correct itself.

The system was tested in a maximal power consumption configuration and various functionality that reduces power consumption was implemented, central of which being the programming model: the firmware is entirely *interrupt-driven*, and the system stays in low-power sleep mode at any point outside of processing data. The cumulative reduction in power consumption from all the techniques implemented was a hefty 45,645% from the maximum measured amount.

A secondary goal was to explore the feasibility of using register-level programming techniques over higher-level ones (e.g HAL) when designing firmware for a commercial device. It can be concluded that while a commercial system can be implemented using register-level programming with success, this approach has several disadvantages, namely: the likelihood that the implementation is going to take more time, the lesser reusability of existing code, and the higher level of skill demanded of the programmer.

Possible improvements to the end result directly correlate knowledge of the hardware at hand: when taking a register-level approach, fine-tuning each module is possible, and perhaps even necessary, depending on the application type. Experience in working close to the hardware level is therefore invaluable for future projects: for example, in an otherwise high-level project, key modules could be implemented with register-level code, gaining noticeable efficiency over the alternative implementation, thus creating a better product than the competition.

References

1. Noergaard, Tammy. Embedded Systems Architecture – A Comprehensive Guide for Engineers and Programmers, Oxford, UK: Elsevier; 2005.

2. Jiming, S., Jones M., Reinauer, S., Zimmer, V.. Embedded Firmware Solutions: Development Best Practices for the Internet of Things. New York, USA: Apress Media LLC; 2015.

3. Broekman, Bart, Notenboom, Edward, Testing Embedded Software. Harlow, UK: Addison-Wesley; 2003.

4. Yiu, Joseph. The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors. Oxford, UK: Elsevier; 2015.

5. STMicroelectronics, STM32L0x3 Reference Manual RM0367 [online]. URL: http://www.st.com/content/ccc/resource/technical/document/reference_manual/2f/b9/c6/34/28/29/42/d2/DM00095744.pdf/files/DM00095744.pdf/jcr:content/translations/en.DM00095744.pdf accessed 28 April 2018.

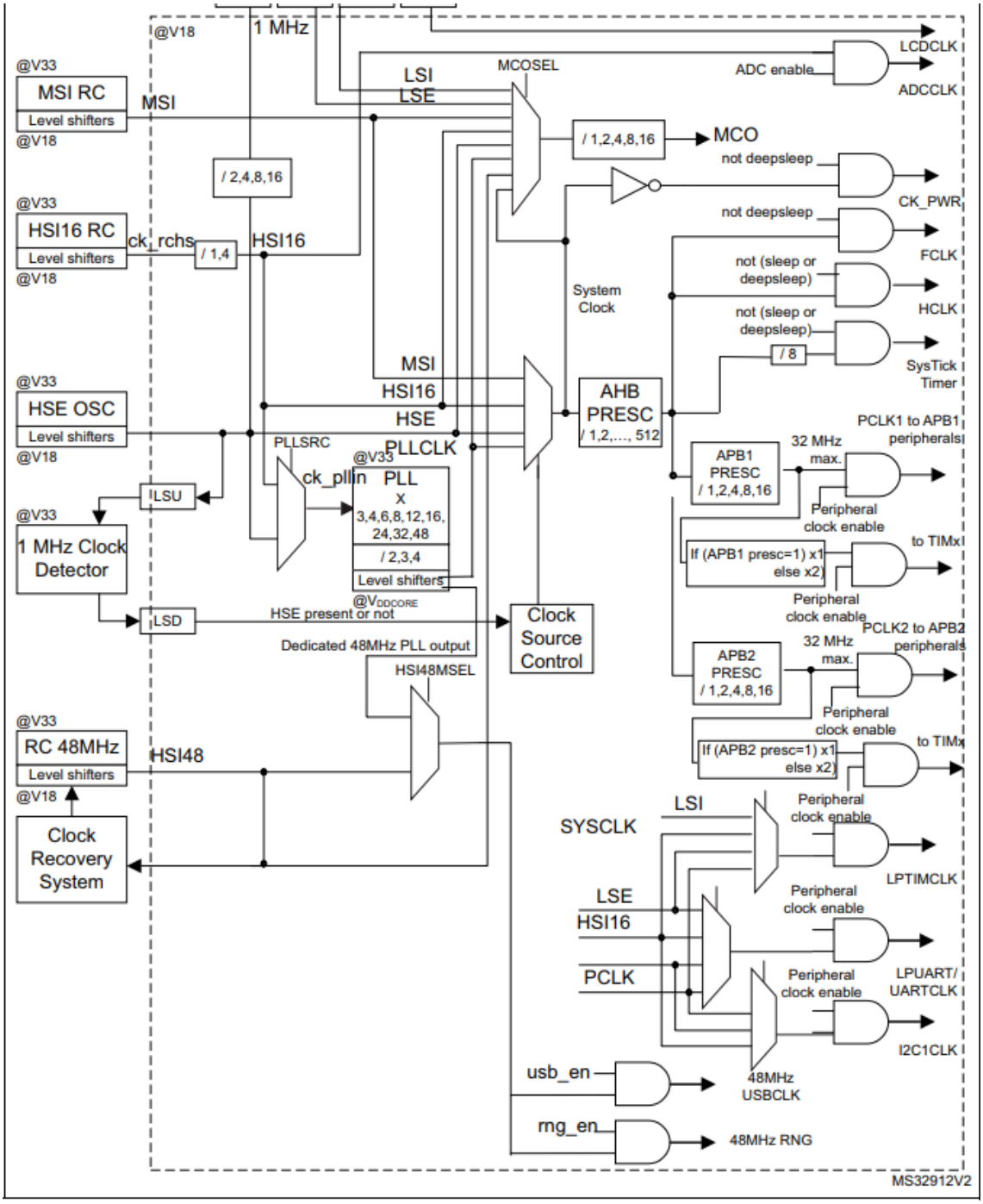
6. STMicroelectronics, STM32L0xx ultra-low power features overview, AN4445 [online].
URL:http://www.st.com/content/ccc/resource/technical/document/application_note/27/58/8e/81/79/fb/4f/ac/DM00108286.pdf/files/DM00108286.pdf/jcr:content/translations/en.DM00108286.pdf , accessed 28 April 2018.

7. STMicroelectronics, STM32L053R8 Datasheet [online]. URL:
http://www.st.com/content/ccc/resource/technical/document/datasheet/8a/f4/9d/d7/61/1b/46/b4/DM00105960.pdf/files/DM00105960.pdf/jcr:content/translations/en.DM00105960.pdf

8. Harvey, A.F. DMA Fundamentals on Various PC Platforms [online]. National Instruments; April 1991. URL: http://ceng2.ktu.edu.tr/~cevhers/ders_materyal/bil311_bilgisayar_mimarisi/supplementary_docs/ni_dma.pdf , accessed 10 April 2018.

9. ARM, MDK-ARM Compiler Optimizations, Application note 202 [online] URL: http://www.keil.com/appnotes/files/apnt202.pdf , accessed 7 May 2018.

STM32L053 Clock tree



Project test setup